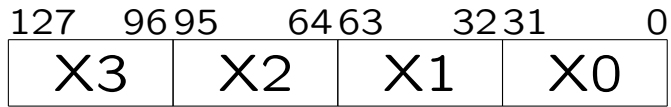


# 1. SIMD Vectorization

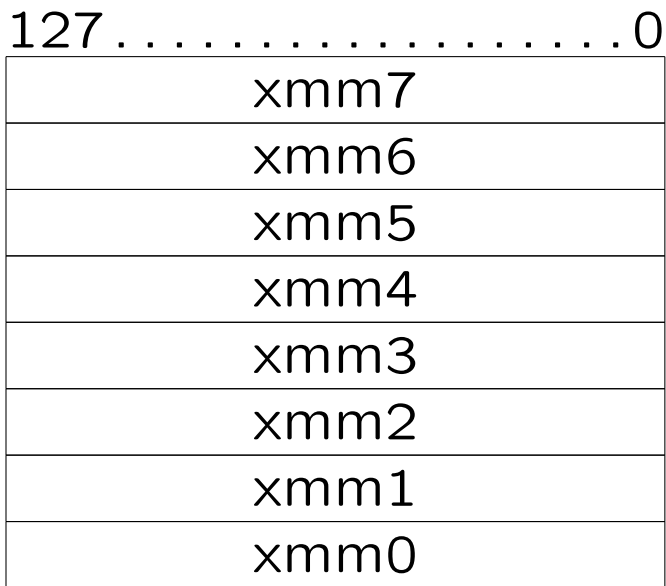
**Single Instruction Multiple Data:** one instruction operates on a vector of data

- For x86 and PPC, vector length is 128 bits / 16 bytes
- Therefore, can operate on 4 single precision values at a time
  - Must unroll loop by  $\geq 4$  to apply!
- We will first discuss SSE (SSE1) for single precision
- x86 SIMD history:
  1. MMX (64bit int ops aliased to x87 regs)
  2. 3DNow! (AMD floats, non-IEEE)
  3. **SSE/SSE1** : SIMD for floats
  4. SSE2 : SIMD for doubles (and integers)
  5. SSE3 : SIMD for complex & cleanup

## 2. Vector Registers (SSE1)



**single prec fp words**



**Vector Registers**

- Register file, not stack
- Each vector reg contains 4 32 bit (single precision) values
- All `xmm` registers *caller*-saved
- `ST(0)` still return value for functions!  
→ Must write to mem, `flds`
- Can read/write only `X0` directly to scalar  
→ Must shuffle vectors around, and reduce to scalar in `X0`
- Same regs used by SSE1/2/3
- Not aliased with `ST(x)`  
→ So far, VPU is overlapped with FPU

### 3. Common SSE1 Data Movement Instructions

Mnemonic	Operands	Action
movaps	m16rs, rd	rd[0:3] = m16rs[0:3]
movaps	rs, m16rd	m16rd[0:3] = rs[0:3]
movups	mrs, rd	rd[0:3] = mrs[0:3]
movups	rs, mrd	mrd[0:3] = rs[0:3]
movss	ms, rd	rd[0] = ms[0]; rd[1:3] = 0.0
movss	rs, mrd	rmd[0] = rs[0]
movss	rs, rd	rd[0] = rs[0]
movlps	rs, md	m[0:1] = rs[0:1]
movlps	ms, rd	rs[0:1] = m[0:1]
movhps	rs, md	m[0:1] = rs[2:3]
movhps	ms, rd	rs[2:3] = m[0:1]
movntps	rs, m16	m16[0:3] = rs[0:3], do not cache m16
movmskps	xmm, ireg	Copy sign bits of xmm's floats to low 4 bits of ireg, zero rest of ireg

- **m16**: 16-byte (vector word) aligned memory address

## 4. SSE1 Permute Operations

Mnemonic	Operands	Action
Permute Instructions		
<code>movhlps</code>	<code>rs, rd</code>	<code>rd[0:1] = rs[2:3]</code>
<code>movlhps</code>	<code>rs, rd</code>	<code>rd[2:3] = rs[0:1]</code>
<code>unpcklps</code>	<code>mrs, rd</code>	<code>rd[0]=rd[0]; rd[1]=mrs[0]; rd[2]=rd[1]; rd[3]=mrs[2]</code>
<code>unpckhps</code>	<code>mrs, rd</code>	<code>rd[0]=rd[2]; rd[1]=mrs[2]; rd[2]=rd[3]; rd[3]=mrs[3]</code>
<code>shufps</code>	<code>imm8, rs, rd</code>	<code>imm8 = dd:cc:bb:aa (binary); rd[0]=rd[aa] rd[1]=rd[bb]; rd[2]=rs[cc]; rd[3]=rs[dd]</code>

## 5. SSE1 Computational Operations

Mnemonic	Operands	Action
andps	m16rs, rd	$rd[0:3] \&= m16rs[0:3]$
andnps	m16rs, rd	$rd[0:3] = (\sim rd[0:3]) \& m16rs[0:3]$
orps	m16rs, rd	$rd[0:3]  = m16rs[0:3]$
xorps	m16rs, rd	$rd[0:3] \hat{=} m16rs[0:3]$ (zero!)
addps	m16rs, rd	$rd[0:3] += m16rs[0:3]$
addss	mrs, rd	$rd[0] += mrs[0]$
subps	m16rs, rd	$rd[0:3] -= m16rs[0:3]$
subss	mrs, rd	$rd[0] -= mrs[0]$
mulps	m16rs, rd	$rd[0:3] *= m16rs[0:3]$
mulss	mrs, rd	$rd[0] *= m16rs[0]$
divps	m16rs, rd	$rd[0:3] /= m16rs[0:3]$
divss	mrs, rd	$rd[0] /= m16rs[0]$
maxps	m16rs, rd	$rd[0:3] = \text{MAX}(rd[0:3], m16rs[0:3])$
maxss	mrs, rd	$rd[0] += \text{MAX}(rd[0], m16rs[0])$
minps	m16rs, rd	$rd[0:3] = \text{MIN}(rd[0:3], m16rs[0:3])$
minss	mrs, rd	$rd[0] += \text{MIN}(rd[0], m16rs[0])$

## 6. SSE1/3DNow! Prefetch Operations

Mnemonic	Operands	Action
prefetchnta	mem	prefetch cache line containing *(mem) to lowest level of cache, hint not heavily reused
prefetcht0	mem	prefetch cache line containing *(mem) to all cache levels
prefetcht1	mem	prefetch cache line containing *(mem) to second level of cache
prefetcht2	mem	prefetch cache line containing *(mem) to third level of cache
prefetchw	mem	<b>(3DNow!)</b> prefetch cache line containing *(mem) with hint that you will write to it

- Fetches cache line at time.
- **0** : L1 on PIII, Athlon & Opteron; L2 on P4.
- **nta** : L1 on PIII, Athlon & Opteron; L2 on P4.
- **1** : L2 on PIII and P4; L1 on Athlon & Opteron.
- **2** : L2 on PIII and P4; L1 on Athlon & Opteron.
- **w** : L1 on Athlon & Opteron.

## 7. Vector Comparisons

Mnem	Operands	Action
cmpps	imm8, m16rs, rd	rd ← all 1s if comparison is true, else all zeros
cmpXXps	m16rs, rd	rd ← all 1s if comparison is true, else all zeros
movmskps	xmm, ireg	Copy sign bits of xmm's floats to low 4 bits of ireg, zero rest of ireg
comisd	mrs, rd	set ICC as shown in table on right

COMP	XX	imm8	NaN
rd = m16rs	eq	0	0
rd < m16rs	lt	1	0
rd ≤ m16rs	le	2	0
rd or m16rs is NaN	unord	3	1
rd ≠ m16rs	neq	4	1
rd ⋈ m16rs	nlt	5	1
rd ⋈ m16rs	nle	6	1

⇒ Use movmskps to get to ireg

⇒ Then use bt and test to branch

rd[0] > mrs[0]	ZF=0, PF=0, CF=0
rd[0] < mrs[0]	ZF=0, PF=0, CF=1
rd[0] = mrs[0]	ZF=1, PF=0, CF=0
NaN	ZF=1, PF=1, CF=1

**comiss result**

## 8. Getting 16-byte Frame Alignment

- For vector temps, 16-byte aligned more efficient
- Frame only 4-byte aligned
- Must manually adjust:
  - Stack grows down, so just zero 4 least sig bits
  - Must keep track of original SP!

```
movl %esp, %eax # save orig SP
subl $FSIZE, %esp
andw $FFF0, %sp
movl %eax, (%esp)
.....
```

DONE:

```
movl (%esp), %esp
ret
```

# 9. Simple SAXPY in x86 SSE Assembly

```
#define ralp    %xmm0
#define rX     %xmm1
#define rY     %xmm2
#define N      %eax
#define X      %edx
#define Y      %ecx
#define II     %edi
#define FSIZE  8

.text
.global ATL_UAXPY
ATL_UAXPY:
    subl    $FSIZE, %esp
    movl    %edi, (%esp)
    movl    FSIZE+4(%esp), N
    movss   FSIZE+8(%esp), ralp    # ralp = {0,0,0,alp}
    shufps  $0x00, ralp, ralp    # ralp = {al,al,al,al}
    movl    FSIZE+12(%esp), X
    movl    FSIZE+20(%esp), Y
# for (i=(N/4)*4; i; i--)
#     Y[N4-i] = alpha * X[N4-i]
    cmpl   $4, N
    jl     CLEANTST
    movl   N, II
    shr   $2, II
    shl   $2, II    # II = (N/4)*4
    lea   (X,II,4), X    # X += N4;
    lea   (Y,II,4), Y
    sub   II, N
    neg   II

LOOP4:
    movups (X,II,4), rX
    movups (Y,II,4), rY
    mulps  ralp, rX
    addps  rX, rY
    movups rY, (Y,II,4)
    addl   $4, II
    jnz    LOOP4
CLEANTST:
    cmpl   $0, N
    jnz    CLEANUP
#
# Epilogue: restore regs and return
#
    movl   (%esp), %edi
    addl   $FSIZE, %esp
    ret
CLEANUP:
    lea   (X, N, 4), X
    lea   (Y, N, 4), Y
    movl  N, II
    neg   II
CULOOK:
    movss (X,II,4), rX
    mulss ralp, rX
    addss (Y,II,4), rX
    movss rX, (Y,II,4)
    inc   II
    jnz   CULOOK
    jmp   DONE
```

## 10. Unsimple SSE SASUM : alignment & other annoyances

```
#define X      %ecx
#define N      %edx
#define N_b    %dl
#define Nr     %eax
#define Nr_b   %al
#define JTARG  %ebx
#define absval %xmm0
#define asum   %xmm1
#define rX     %xmm2
#define FSIZE  24
# float ATL_UASUM(int N, float *X, int incX)
.global ATL_UASUM
ATL_UASUM:
    movl    %esp, %eax    # save orig SP
    subl   $FSIZE, %esp
    andw   $0xFFF0, %sp  # sp now 16-byte aligned
    movl   %eax, 16(%esp)
    movl   %ebx, 20(%esp)
# construct ~(-0.0) in absval
    movl   $0x7FFFFFFF, %ecx
    movl   %ecx, (%esp)
    movss  (%esp), absval
    shufps $0x00, absval, absval
    movaps absval, (%esp)
    movl   4(%eax), N
    movl   8(%eax), X
    xorps  asum, asum

# Nr = (((char*)X)+15)/16)*16 - X
    movl   $ALIGNED, JTARG
    lea   15(X), Nr
    andb  $0xF0, Nr_b
    subl  X, Nr
    jnz   FORCE_ALIGN
.local  ALIGNED
ALIGNED:
    cmp   $0, N
    je    DONE
    mov   N, Nr
    andb  $0xFC, Nr_b    # N = (N/4)*4
    subl  N, Nr          # Nr = N-(N/4)*4
    movl  $DONE, JTARG
    cmp   $0, N
    je    LOOP1
.local  LOOP4
LOOP4:
    movaps (X), rX      # rX = {x3, x2, x1, x0}
    andps  absval, rX
    addps  rX, asum
    addl   $16, X
    subl  $4, N
    jnz   LOOP4
    cmp   $0, Nr
    jne   LOOP1
.local  DONE
```

## 11. Unsimple SSE SASUM : continued

DONE:

```
# reduce asum to scalar          asum = {s3, s2, s1, s0}
movhps  asum, rX                 # rX   = { X,  X, s3, s2}
addps   rX, asum                 # asum = { X,  X, s1+s3, s0+s2}
movaps  asum, rX
shufps  $0b11100101, rX, rX     # rX   = {X, X, X, s1+s3}
addss   rX, asum                 # asum = {X,X,X, s0+s2+s1+s3}
movl    (%esp), %ebx
# put asum in ST(0) for return value
movss   asum, (%esp)
flds    (%esp)
# Restore int regs & return
movl    20(%esp), %ebx
movl    16(%esp), %esp
ret
.local  FORCE_ALIGN
FORCE_ALIGN:
shrl    $2, Nr   # Nr = (Xa-X)/sizeof(float)
cmp     N, Nr
cmova   N, Nr   # Nr = MIN(N,Nr)
sub     Nr, N
.local  LOOP1
LOOP1:
movss   (X), rX
andps   absval, rX
addss   rX, asum
add     $4, X
dec     Nr
jnz     LOOP1
jmp     JTARG
```

For best perf, must:

- Separate aligned cases
  - Peel for only one array
  - Peel for alignment on most used array, if others are mutually misaligned, used unaligned access for them
- Opt each loop separately
- Store all vector temp aligned
  - Must manually align SP